COP-4610 Homework Assignment 3

Date of assignment: October 13, 2017
Deadline: November 2, 2017 at 23:55

This design of this assignment is accredited to Prof. Jinpeng Wei. Modifications were made to fit with the current course whenever appropriate.

Until this assignment is formally assigned at the date of assignment, it may be subject to minor modifications.

Objectives:

This assignment includes several important topics on process management. You will program with the widely used pthread programming interface. The program will be done in user space (pthread is part of the GNU C library). You will also little bit of work inside the kernel to be familiar with kernel data structures related to processes and threads.

Description:

In this assignment, you will be working with the "threads"' subsystem of Linux. This is the part of Linux that supports multiple concurrent activities within the kernel. In the exercises below, you will write a simple program that creates multiple threads, you will demonstrate the problems that arise when multiple threads perform unsynchronized access to shared data, and you will rectify these problems by introducing synchronization (in the form of Pthreads mutex) into the code. You will also add a new system call, which allows you to inspect process and thread information from the kernel.

Part 1: Simple Multithreaded Programming

The purpose of this exercise is for you to exercise using the threads primitives provided by pthreads [1], and to demonstrate what happens if concurrently executing threads modify shared variables without proper synchronization. Then you will use the mutex synchronization primitives in pthreads to achieve proper synchronization.

Step 1.1: Simple Multithreaded Programming without Synchronization

First, you need to write a program using the pthread library that forks a number of threads each executes the loop in the `SimpleThread` function below. The number of threads is a command line parameter of your program. All the threads modify a shared variable `SharedVariable` and display its value within and after the loop.

```
int SharedVariable = 0;

void SimpleThread(int which) {
      int num, val;
      for(num = 0; num < 20; num++) {
            if (random() > RAND_MAX / 2)
                  usleep(10);
            val = SharedVariable;
            printf("*** thread %d sees value %d\n", which, val);
            SharedVariable = val + 1;
      }
      val = SharedVariable;
      printf("Thread %d sees final value %d\n", which, val);
}
```

Your program must validate the command line parameter to make sure that it is a number, not arbitrary garbage. Also, your program must be able to run properly with any reasonable number of threads (e.g., 200).

Try your program with the command line parameter set to 1, 2, 3, 4, and 5. Analyze and explain the results. Put your explanation in your project report.

Step 1.2: Simple Threads Programming with Proper Synchronization

Modify your program by introducing pthread mutex variables, so that accesses to the shared variable are properly synchronized. Try your synchronized version with the command line parameter set to 1, 2, 3, 4, and 5.

Accesses to the shared variables are properly synchronized if: (i) each iteration of the loop in `SimpleThread()` increments the variable by exactly one, and (ii) each thread sees the same final value. It is necessary to use a pthread barrier [2] in order to allow all threads to wait for the last to exit the loop.

You must surround all of your synchronization-related changes with preprocessor commands, so that we can easily compile and get the version of your program developed in Step 1.1. For example:

```
for(num = 0; num < 20; num++) {
#ifdef PTHREAD _SYNC
      /* put your synchronization-related code here */
#endif
      val = SharedVariable;
      printf("*** thread %d sees value %d\n", which, val);
      SharedVariable = val + 1;
      .......
}
```

One acceptable output of your program is (assuming 4 threads):

```
*** thread 0 sees value 0
*** thread 0 sees value 1
*** thread 0 sees value 2
*** thread 0 sees value 3
*** thread 0 sees value 4
*** thread 1 sees value 5
*** thread 1 sees value 6
*** thread 1 sees value 7
*** thread 1 sees value 8
*** thread 1 sees value 9
*** thread 2 sees value 10
*** thread 2 sees value 11
*** thread 2 sees value 12
*** thread 3 sees value 13
*** thread 3 sees value 14
*** thread 3 sees value 15
*** thread 3 sees value 16
*** thread 3 sees value 17
*** thread 2 sees value 18
*** thread 2 sees value 19
......
Thread 0 sees final value 80
Thread 2 sees final value 80
Thread 1 sees final value 80
Thread 3 sees final value 80
```

Part 2: Inspect Tasks in Kernel

Create a new system call that can dump information about all processes and threads in the system. For each process or thread, you need to print its command line, state (running, waiting, etc), process id, and its parent process's id. The print-out can be done using `printk`. You should use the same method as in the previous assignment to create this new system call. The name of the system call should be `sys_print_tasks_firstname_lastname`.

Modify your program so that it calls your new system call after all threads have been created but before they are terminated. Use `dmesg` to verify that information about all threads of your program can be dumped inside your system call. For example, if your program creates 3 threads, dmesg output should show all 3 of them.

Hint: use the macros `for_each_process` and `while_each_thread` to iterate through the list of process control blocks in Linux. More information can be found in references [4] and [5].

Deliverables

- A README file, which describes how we can compile and run your code;
- Your source code, including a Makefile.
    - Do not submit your binary code---your grader will compile your code.
    - No grade will be given if your code fails to compile. Your code should not produce anything else other than the required information in the output file;
    - Provide sufficient comments in your code to help the TA understand your code; this is important in case you want to get partial credit when your submitted code does not work properly;
- A patch file that represents the changes that you made to the vanilla 2.6.36 kernel to implement the new system call. (It is okay to include the other system call that you had your previous assignment);
- Your report, which should discuss the output of your program without pthread synchronization and the one with Pthread synchronization, as well as the reason for the difference. In addition, the report should also include the output of dmesg that shows the process and thread information.
- Compress all the above into a single file and upload it into Moodle.

Grading Criteria

- Simple multithreaded program without synchronization (30%)
- Simple multithreaded program with synchronization (20%)
- System call for inspecting processes and threads (40%)
- Report (10%)

References

[1] POSIX Threads Programming: https://computing.llnl.gov/tutorials/pthreads/
[2] Pthreads Primer: http://pages.cs.wisc.edu/~travitch/pthreads_primer.html
[3] POSIX thread (pthread) libraries:
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
[4]
http://stackoverflow.com/questions/19208487/kernel-module-that-iterates-over-all-tasks-using-depth-first-tree

[5]
http://stackoverflow.com/questions/8457890/kernel-how-to-find-all-threads-from-a-processs-task-struct

This is individual project. You should work on this project alone.

The deadline for this assignment is at 23:55 on the due date. Late submission will incur penalties as described in the syllabus.